Noname manuscript No.
(will be inserted by the editor)

Hiding a fault enabled virus through code construction

Samiya Hamadouche · Jean-Louis Lanet · Mohamed Mezghiche

Received: date / Accepted: date

Abstract Smart cards are very secure devices designed to execute applications and store confidential data. Therefore, they become the target of many hardware and software attacks that aim to bypass their embedded security mechanisms in order to gain access to the sensitive stored data. Recently, a new kind of attacks called combined attacks has appeared. They aim to induce perturbations in the application's execution environment. Thus, correct and legitimate application can be dynamically modified to become a hostile one after being loaded in the card using a fault injection. In this paper, we treat the problem from another angle: how to design an innocent looking code in such a way that it becomes intentionally hostile after being activated by a fault injection? We present an original approach of backward code construction based on constraints satisfaction and a tree traversal algorithm. After that, we propose a way to optimize the search process by introducing heuristics for a faster convergence towards more realistic solutions.We implement this approach in a *Trace Generator* tool; thereafter evaluate its capacity to generate the required solutions while giving a proof-of-concept of the code desynchronization technique.

Keywords Java Card bytecode · fault injection · Constraint Satisfaction Problem · tree traversal · backward code construction · code desynchronization

S.Hamadouche, M.Mezghiche
LIMOSE Laboratory, Faculty of science, University M'Hamed Bougara of Boumerdes, Avenue de l'Indpendance, 35000 Algeria
E-mail: hamadouche.samiya@univ-boumerdes.dz, mohamed.mezghiche@univ-boumerdes.dz

JL.Lanet
INRIA, LHS-PEC 263 Avenue Gnral Leclerc, 35042 Rennes, France
E-mail: jean-louis.lanet@inria.fr

1 Introduction

Embedded code is present in billions of devices all over the world. Many of these devices handle privileged information, making their security an important concern. The most used example is smart card present in our day to day life e.g. identity, mobile, banking or e-passport applications. Therefore, they have become the target of many attacks that aim to circumvent the embedded security mechanisms to gain access the sensitive data they contain even take control of the system through unauthorized access.

The system's code is stored in the ROM (Read Only Memory) and it can be affected by a fault attack while it transits on the bus but this remains a transient fault which is more difficult to be exploited by the attacker. However, the permanent fault is the most valuable. This can occur when the applicative code stored in the NVM (Non Volatile Memory) is attacked. But some of the recent smart cards have also the system code stored in FLASH memory which may be subjected to a permanent fault, too. In both cases, modifying the stored code can change the behaviour of the application leading to a potential threatening application. Such a modified application, not detected by the embedded countermeasures, is defined as a *mutant*.

Until now, researchers have focused on the capacity to modify the behaviour of the system. We want to focus on the capacity for a hostile developer to design a code with a dual correct semantics. The normal semantics that behaves as expected, and a second one which can be activated on demand by the attacker with a fault attack. Such behaviour can be considered as an obfuscation technique which aims to hide a payload into a regular code.

Program obfuscation is a major method for software intellectual property protection. Furthermore, it is broadly used by malware authors to hide their malicious code and thus evade detection. It transforms a program into a new version which is semantically equivalent with the original one but much harder to understand and analyse. In our case, we aim to dissimulate a code, too. However, the challenge and originality of our work lies on the capacity to have a same code owning at least two semantics, i.e. a polymorphic code. In other words, we have to hide the malicious code inside a well-typed program so that the resulting program is semantically correct even after the fault injection. For that, we have to insert one or more instructions just before the code to hide in such a way that some constraints are verified in order to avoid some embedded countermeasures. Moreover, the beginning of the code to hide must be concealed in the added instructions. That is to say, it is necessary to hide the real operation as a part of the operands of the preceding instruction. This mechanism is called *code desynchronization*.

The rest of the paper is structured as follows. Section 2 recalls background information by answering the questions *how to hide a code?* and *how can we*

activate it? in a general context in order to position our work. Sections 3 and 4 present the theorical foundation and formalisation of the treated problem. Section 5 introduces the basic elements of the proposed approach of code sequence construction which is discussed in section 6, and section 7 for its optimisation through heuristics use. Section 8 exposes the implemented tool whose experimental results are explained through a practical case study in section 9. The paper is concluded in section 10.

2 Context

2.1 Code obfuscation

Obfuscation consists in applying code transformations in order to obtain new program versions that are harder to understand and analyze manually or with automatic tools, while preserving its semantics. As obfuscation has been studied for more than two decades, several surveys treating different aspects are available. The groundbreaking study was presented in 1997 by Collberg et al. [19] who proposed a detailed taxonomy of obfuscation transformations that becomes the basis of almost next studies proposed in the literature. Among many other researches, we can cite Drape et al. [20] who surveyed several obfuscation techniques via layout tranformation, control-flow tranformation, data tranformation, language dependent transformations, *etc*; Balakrishnan and Schulze [3] surveyed several major obfuscation approaches for both benign and malicious codes; Xu et al. [60] surveyed the existing approaches for code-oriented obfuscation and model-oriented obfuscation with a comparative study of the two classes. Obfuscation is employed in various domains and several transformations (general or specific) can be applied on different program's levels. In the following, we briefly present an overview about these aspects.

2.1.1 Usages

Code obfuscation is widely employed in practice. The existing techniques are in general used for one or more of the following purposes [22]:

- To protect intellectual property from rivals by making reverse-engineering very difficult.
- To protect Digital Rights Management of multimedia resources in order to reduce piracy.
- Developers perform obfuscation on their applications to make them more compact and thus faster.
- Malware authors use obfuscation to hide their creations from anti-malware scanners and deep-analysis for a longer duration, so that it can propagate and infect more and more devices. Prevent or at least delay human analysts or automatic analysis engines from figuring out the intention of malicious code.

2.1.2 Obfuscation quality

Collberg et al. [19] and Low [38] propose to evaluate obfuscation transformations according to four quality metrics:

- *Potency* measures how much more difficult the obfuscated code is to understand than the original code.
- *Resilience* measures how well a transformation holds up under attack from an automatic deobfuscator.
- *Stealth* determines how well the obfuscated code blends with the rest of the program.
- execution *cost* concerns the time/space penalty (i.e. computation overhead) that is added to the original program;

Therefore, a transformation quality is defined as a combination of the previous metrics to express how suitable it is.

2.1.3 Obfuscation levels

Obfuscation transformations can be applied on different representations of a program, mainly the *source code*, the *Intermediate Representation (IR)* or the *assembly language*. Furthermore, combinations are possible as transformations can be applied sequentially on a piece of code at different levels during the code compilation process [16].

Source code level. An obfuscation at this level is called a source-to-source obfuscation. It exploits specificities of the input programming language. As the obfuscation step is taking place before the compilation, it is easier to integrate it into an existing compilation chain. It might be easier to apply certain transformation techniques on a high level, as source code is richer compared to binary code [33]. In Madou et al. [39], authors perform high-level transformations, compile the target application, and observe the protection in the binary. Supported by empirical results, they conclude that several software protection techniques survive compiler transformations. Hence, it is not always necessary to apply additional binary transformation steps to applications designed in source code. Collberg et al. have extensively described techniques available for source code obfuscation in [19], [18]. Examples of such obfuscators are DashO[1], DexProtector[2] or ProGuard[3] for Java and Dotfuscator[4] for .NET.

Intermediate Representation level. Intermediate Representation (IR) is designed to be independent of any source or target language. Thus, obfuscators working at this level are more general than source-to-source obfuscators, and could treat programs from different source languages. However, the integration

[1] https://www.preemptive.com/products/dasho/overview
[2] http://dexprotector.com
[3] http://proguard.sourceforge.net/
[4] http://www.preemptive.com/products/dotfuscator/

is more difficult as it requires the obfuscator to be added to the existing compilation toolchain [21]. Examples of such obfuscators are Obfuscator-LLVM[5] that works on the LLVM Intermediate Representation (IR) code [31], or Epona[6] a commercial obfuscator developed by Quarkslab.

Assembly Language level. The assembly level presents a major loss of information compared to the IR and source levels, thus it is very difficult to implement a general obfuscator working only on assembly [21]. One technique consists of applying protection by virtualization directly on binary programs. The protected code then runs on a virtual CPU different from standard CPUs. VMProtect[7] is a commercial obfuscator implementing virtualization. Another technique related with low level obfuscation is Instruction Set Randomization (ISR) [33], [8]. A unique execution environment to the running process is created. In other words, a new instruction set is created for each process executing within a system. Therefore, the attacker does not know the language being used and cannot communicate with the machine.

2.1.4 Classical obfuscation techniques

The classification of obfuscation techniques as presented by Collberg et al. [19] is based on the target of transformation. In other words, what is transformed and how the transformation is applied. The main three defined categories are: *layout, control-flow* and *data* obfuscations. In the following, we give a brief overview about each category. Moreover, figure 1 resumes the most discussed classical obfuscation techniques that fall in the cited categories. For more details about those techniques and additional examples, the reader could refer to works such as [19], [20], [29].

Layout transformations. They target the programs layout structure by changing its look while keeping its semantics intact. By reducing the amount of information for the human reader, the reverse engineering becomes harder. Layout transformations are done for example through renaming the identifiers, removing the comments and information about debugging, and source code formatting.

Control flow transformations. They aim to increase the obscurity of programs control flow. There exists a large body of research on control flow obfuscation techniques. Principally, they affect the *aggregation, ordering* or *computations* of the control flow [19]. Aggregation transformations split computations that are logically related and merge independent computations. Control ordering transformations reorder the code blocks, loops and expressions while preserving their dependencies. Computation transformations insert new code

[5] https://github.com/obfuscator-llvm/obfuscator/
[6] https://epona.quarkslab.com/
[7] http://vmpsoft.com/

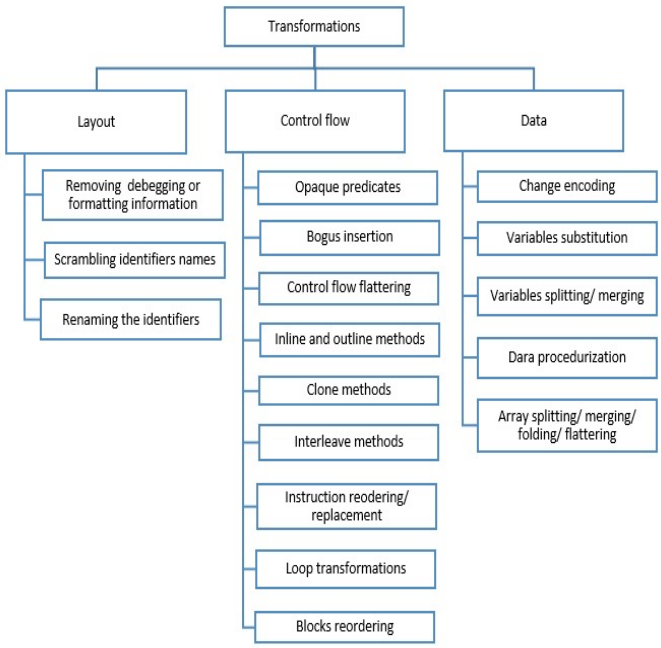

Fig. 1: Examples of obfuscation techniques

or make algorithmic changes to source application. Many of the control flow transformations rely on the notion of *opaque predicates* introduced by Collberg et al. in [19]. They defined an opaque predicate as the predicate (a boolean expression) whose outcome is known during obfuscation time but is difficult to deduce by static program analysis.

Data transformations. They aim to obfuscate the data and the data structures that a program may use. The values taken during execution are concealed, as well as the information that can be inferred from the data organization and interactions. Various approaches have been used for this aim. According to Collberg et al. [19] those transformations concern *storage and encoding* that change representation of data, *aggregation* which splits/merges data and *ordering* that permutes items in existing data structures.

2.1.5 Specific obfuscation techniques: malware application

The malware developers use obfuscation techniques to conceal the malicious code in order to bypass the malware detection system. The camouflage in malware has an exponential growth over the years from simple encryption to complex polymorphic and metamorphic malware [23], [52]. These two malware are the main types of malicious code using obfuscation techniques to hide

themselves from virus scanners. They rely on techniques that change their code signature at each infection generation [3].
An encrypted virus simply encrypts its body and then attaches to itself a decryption key which changes from generation to generation. A polymorphic virus [3] also encrypts its body, however, it changes its decryption algorithm each infection. Therefore, it changes the part of the signature that scanners could detect. Metamorphic viruses [3], like polymorphic viruses encrypt themselves to hide their signatures from virus scanners. However, metamorphic viruses can change their source code and then recompile themselves if compilers are available on the victim machine. Unlike polymorphic viruses, metamorphic viruses do not decrypt themselves in memory to propagate and infect hosts. In fact, a metamorphic virus never reveals its entire virus body at once, making it almost impossible for a simple signature-matching scanner to detect it.
Below are the most common obfuscation techniques, among many others, that are particularly used to hide malware. More details and practical examples can be found in [11], [61], [48], [52], [51].

Junk Code Insertion. It involves placing ineffective instructions in the structure of the virus to change its appearance without affecting its behaviour. These new instructions can be for example [11], [2]: an instruction that does not change the content of CPU registers or memory and is equal to no-operation (NOP); an instruction that probably changes the status of the machine or the content of memory or CPU registers but its effect is cancelled by another one before affecting the programs result (e.g. pop/push).

Dead Code Insertion. It refers to insertion of unreachable code blocks and thus never get executed. Inclusion of such code can make the analysis of a program more time consuming as it increases the amount of code that has to be analysed. For making the identification of dead code more difficult, opaque predicates [19] that always resolve to either true or false can be used.

Instruction substitution. It replaces some instructions in the original code with other equivalent ones. As this technique requires the use of a library of equivalent instructions (which is unavailable), the signature of the original code can greatly be changed.

Register Substitution. It aims to substitute registers in different instances of the virus. The overall functionality is preserved while the programming structure of the virus changes. With this method, the virus tries to defeat the string signature detection.

Code Transposition. It reorders the instructions sequence of an original code without having any impact on its behaviour. Through this rearranging process, as different combinations of instructions are applied, the structure of the code looks dissimilar in various generations. For example, new genera-

tions can be created by choosing and reordering the independent instructions that have no impact on another one. Another example is to reorder the program instructions, but the execution flow is still kept using unconditional or conditional branches.

2.2 Fault injection attacks

2.2.1 The concept

Fault Injection (FI) is an old research discipline [28], [1], [56], [40], which has its origin in fault tolerance systems mainly from aerospace domain. Researchers brought to the fore that cosmic rays can flip single bits in the memory of an electronic device. The impact of physical phenomena on embedded systems has been widely studied by the scientific community, with a particular interest on secure systems [5]. Several types of fault attacks are focused by researchers in the smart card field. Boneh, DeMillo and Lipton have proposed in [10] a new attack against smart cards, called cryptanalysis, in presence of hardware fault. This attack model initially focused on several public-key cryptographic algorithms like the RSA. This has led to numerous forms of hardware attacks against smart cards using fault injection.

Faults can be induced into the chip by using physical perturbations in its execution environment. Several techniques have been successfully demonstrated in the literature [4], [32], [47], [62]. Commonly used ones include: *Overclocking* [24], *Clock glitching* [35], *Underfeeding* [7], *Voltage glitching* [54], *Overheating* [30], *Electromagnetic emission (EM)* [50], *Light pulse* [53] and *Laser beam* [25].

The introduced errors can generate different versions of a program by changing some instructions, interpreting operands as instructions, branching to other (or invalid) labels and so on. A fault attack has the ability to physically disturb the smart card chip. At the hardware level, the basic effect is a change in a transistor. At the bit level we distinguish several types of fault: bit-set, bit-flip, bit-reset, stuck-at and random-value. It can manifest as single or multi-bit faults but also whole byte or burst of bytes in memory. The memory cell can be part of internal CPU state, Instruction-Set Architecture (ISA), visible CPU registers, or any other part of the memory hierarchy, including CPU cache, SRAM or main memory DRAM. Mainly, a fault attack permits an attacker to execute a treatment beyond his rights, or to access secret data in the smart card.

2.2.2 Fault Model

It is necessary to provide a model of the possible errors induced by a fault injection in order to evaluate the possible consequences and thus prevent the

occurrence of FI attacks. Fault models commonly considered in the literature mainly depends on three properties, *location*, *fault-type* and *time* [59]. Location denotes where the fault is injected, fault-type denotes which type of fault that is injected and time denotes when the fault is enabled. Sometime, this last attribute is qualified as injection trigger and fault latency. FI models have been already discussed in details in [9], [58]. The different fault models given in descending order in terms of attackers power, are shown in table 1.

Fault model	**Precision**	**Location**	**Timing**	**Fault type**	**Difficulty**
Precise bit error	bit	full control	full control	bsr[8]	++
Precise byte error	byte	full control	full control	bsr, random	+
Unknown byte error	byte	lose control	full control	bsr, random	-
Random error	variable	no control	partial control	random	- -

Table 1: Existing fault models

An attack using the *precise bit error* model had been described by Skorobatov and Anderson in [53]. However, it is not realistic on current smart cards due to the implementation of hardware security on memory (error correction and detection code or memory encryption) of modern components. A widely accepted model corresponds to the *precise byte error* model where an attacker may change one byte at a precise and synchronized time [57].

To illustrate the effect of a fault injection according to the *precise byte error* model, we present the following example (listing 1) which consists in a debit method that belongs to a wallet Java Card applet.

Listing 1: The debit method's code

```
private void debit (APDU apdu)
{
if(pin.isValidated()) {
// make the debit operation
}
else {
ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);}
};
```

In this method, the user's PIN (Personal Identification Number) must be validated prior to the debit operation. The corresponding byte code representation (before and after the fault injection) is given in table 2. An attacker wants to bypass the PIN test. He injects a fault on the cell containing the

[8] bit set or reset

conditional test byte code. Thus, the `ifeq` instruction (byte `0x60`) changes to a `nop` instruction (byte `0x00`). The verification of the PIN code is bypassed, the debit operation is made and an exception is thrown but too late because the attacker will have already achieved his goal.

Byte code before FI		**Byte code after FI**	
Byte	**Byte code**	**Byte**	**Byte code**
00: 18	00: `aload_0`	00: 18	00: `aload_0`
01: 83 00 04	01: `getfield` #4	01: 83 00 04	01: `getfield` #4
04: 8B 00 12	04: `invokevirtual` #18	04: 8B 00 12	04: `invokevirtual` #18
07: 60 00 3B	07: ifeq 59	07: 00	07: nop
10: ...	10: ...	08: 00	08: `nop`
...	...	09: 3B	09: `pop`
56: 70 00 42	56: `goto` 66	10: ...	10: ...
59: 13 63 01	59: `sipush` 25345	...	...
63: 8D 00 0D	63: `invokestatic` #13	56: 70 00 42	56: `goto` 66
66: 7A	66: `return`	59: 13 63 01	59: `sipush` 25345
		63: 8D 00 0D	63: `invokestatic` #13
		66: 7A	66: `return`

Table 2: byte code representation before and after the fault injection

2.2.3 Fault enabled logical attacks

In addition to their use in cryptanalysis, fault attacks can also be used to trigger logical attacks (e.g., control flow hijacking, privilege escalation, subverting memory isolation) on general-purpose processors [44], [49], [54], [45], [34], [15] and smart cards as a special case. As we are interested by this latter, examples from the literature are briefly presented below.

Barbu et al. [6] succeed to bypass the embedded smart card Byte Code Verifier (BCV). The attack consists to install a correct applet containing an unauthorized cast between two different objects. Statically, the applet is compliant with the Java Card security rules. If a laser beam hits the bus in such a way that the cast type check instruction (`checkcast`) is not executed, this applet becomes hostile and can execute any shell code. This type of attack exploits a new method to execute illegal instructions where the physical and logical levels are perturbed. This method succeeds only on some cards and others seem to not be sensitive to this attack.

Bouffard et al. [12], proposed an attack to perturb the applets Control Flow Graph (CFG) with a laser beam injection into the smart cards non-volatile memory. The authors described the attack on a for loop, but it can be extended to other conditional instructions. The Java Card specification [12] defines two instructions to branch at the end of a loop, a `goto` and the `goto_w` instructions. The first one branches with a 1-byte offset and the second one takes 2-byte offset. Since the smart cards memory manager stores the array

data after the memory byte code, a laser fault on the high part of the `goto_w` parameter can shift the backward jump to a forward one. Thus the authors succeeded to execute the contents of an array. Unlike Barbu et al., Bouffard et al. described a persistent attack to execute their shellcode.

Lancia [37] proposed a paradigm for combined attacks that permit to evade the localization precision constraints of the fault injection in order to raise its chances of success. This is possible with creating favorable pattern in persistent memory through instances allocation. The author performed a type confusion to be able to access the arbitrary memory location. The optimal used data patterns are based on the knowledge of memory allocation mechanism. The attack is evaluated using a fault simulator developed by the author.

Bouffard and Lanet [13] presented a generic approach based on a Control Flow Transfer (CFT) attack to modify the Java Card program counter. The attack is based on a type confusion, it abused the BCV verification, using the couple of instructions `jsr/ret`. This allowed them to hide an unreached piece of code, and activate it at runtime using a fault injection.

Mesbah et al. [42] studied the behavior of the Oracle BCV towards unchecked codes and found the way to bypass it. Taking advantage of this breach on the BCV and the understanding of the internal structures of some Java Cards, they demonstrated the ability to load an illegal code (underflow attack). This gives an access to the data system of the frame, and persistently activate any code. Using both a white-box approach and fault injection, a well-formed code can be transformed to an ill-formed one during runtime execution.

2.3 Fault enabled virus

With the emergence of combined attacks, correct and legitimate applications (especially we are interested in Java Card applications) can be dynamically modified after being loaded in the card using a fault injection. Thus, it becomes a hostile application. We have called such an application a *fault enabled virus*. It is a program with two semantics: before and after the fault injection. Both of them are correct i.e. they respect the Oracle specification [46].

Like obfuscation techniques, our work aims to dissimulate a code to make its understanding and analysis more difficult. Nevertheless, the difference is that obfuscation preserves the semantics whereas in our approach we want to hide even the semantics which makes the analysis even more difficult. The originality of our work lies in the fact that we aim to build a program with two correct semantics. In the first one, the hostile behaviour is dissimulated and then recovered in the second one after the fault injection.

The difficulties are related to the possibility to find the right instruction which could have the expected behaviour once it is hit by the laser and to generate the preamble that leads the memory in the state required by the hostile code to be executed. Therefore, our problem can be divided into two complementary parts:

- The first part concerns the construction of the bytecode sequence that links two pieces of code (as shown in figure 2); the beginning of the hostile code and another inoffensive piece of code. This corresponds to the code's preamble.
- The second part concerns what we called the code desynchronization problem. Indeed, the insertion of instructions before the hostile code is not without effect. The byte containing the operation code (opcode) is decoded to determine the number of following bytes corresponding to operands. If a fault injection attack changes the opcode byte, then it may change the number of following bytes used as operands. Thus, a shift in the original instruction flow occurs, the obtained binary code can be interpreted as a new program. However, the original program and the shifted one may, at some point, be the same i.e. the original flow is recovered.

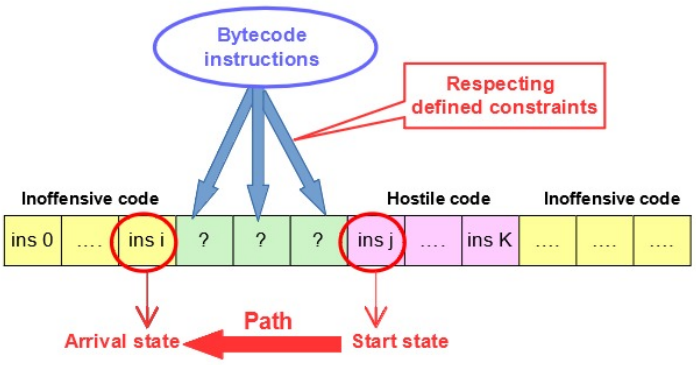

Fig. 2: Bytecode sequence construction problem

Due to the difficulty of the main problem we chose to treat each part separately. In this paper, we are interested only in the first part i.e. how to build the code sequence while respecting the imposed constraints? The second part is briefly treated in section 9 (proof-of-concept) while a more detailed study is left for another future work.

As illustrated by figure 2, we aim to find, among the set of possible bytecode instructions, a sequence of instructions to add in order to rely two given instructions: the beginning of the hostile code and the end of an inoffensive

code fragment. This is done in the opposite direction of the execution (backwards). We must ensure that the insertion of these instructions respects a set of constraints in order to obtain a syntactically and semantically correct program.

The present work is based on the following assumptions:

- The target is Java Card application (this could be extended)
- The code construction is done at the bytecode level and in backward direction
- The fault injection is the trigger of our virus. We are interested in the fault effect not how it is physically injected.
- The chosen fault model is a precise byte error model with unencrypted memory. When the fault occurs, the instruction stored in the target-memory cell is transformed into a `nop` instruction (`0x00`)

Initially, we have demonstrated (proof-of-concept) the possibility to design applications in such a way that they become intentionally hostile after being hit by a laser beam [26]. After that, we stated that this virus construction gets back to a Constraint Satisfaction Problem (CSP)[27]. In the next sections, we will develop this idea by giving a more formal definition of our problem and the adequate solution.

3 Constraint Satisfaction Problems (CSP) : Representation and solving

Constraint programming is a powerful and well-studied paradigm applied to solve combinatorial search problems. It is currently applied with success to many real life problems, such as scheduling, planning, vehicle routing, natural language processing, optimisation problems, molecular biology, resource allocation analysis, synthesis of electronic circuits, network configuration, *etc.* Basically, a constraint satisfaction problem (CSP) is a problem composed of a finite set of variables, each associated to a finite domain, and a set of constraints that restrict the values the variables can simultaneously take [55]. The goal is to assign a value to each variable satisfying all the constraints.

More formally, a CSP is a triple (X, D, C), where:

- $X = \{x_1, x_2, ..., x_n\}$ is the set of variables;
- $D = \{d_1, d_2, ..., d_n\}$ is the set of domains. Each domain is a finite set containing the possible values of the corresponding variable;
- $C = \{c_1, c_2, ..., c_n\}$ is the set of constraints. A constraint is simply a logical relation, among several variables, that restricts the possible values that variables can take. A constraint can be given either explicitly, by enumerating the allowed combinations, or implicitly, e.g. by an algebraic expression.

A solution to a CSP is an assignment of values to all of its variables that satisfies all of its constraints. We may want to find:

- Just one solution, with no preference,
- All solutions,
- An optimal, or at least a good solution, given some objective functions defined in terms of some or all of the variables.

Constraint satisfaction problems are combinatorial in nature. Thus, an algorithm that guarantees to find a solution that satisfies all constraints, assuming that such a solution exists, is enumerative [1]. Therefore, the maximum time taken to complete this procedure grows exponentially with the number of variables. Several different approaches can be applied to solve a CSP. This paper is not intended to provide a survey of constraint satisfaction algorithms. However, we present the most important techniques bellow. More details can be found in [14], [36], [43] and [55].

Generate-and-test (GT) method searches systematically the space of complete assignments i.e. it explores each possible combination of the variable assignments. First, the GT algorithm generates some complete assignment of variables and, then, it tests whether this assignment satisfies all the constraints. If the test fails, i.e. there exists any unsatisfied constraint, then the algorithm tries another complete assignment. The algorithm stops as soon as a complete assignment satisfying all the constraints is found, this is the solution of the problem, or all complete assignments are explored, i.e. the solution does not exist. The number of combinations considered by this method is equal to the size of the Cartesian product of all the variable domains. This is not very efficient because the method generates many wrong assignments of values to variables which are rejected in the testing phase. In addition to that, conflicting instantiations are not considered while generating other assignments.

A more efficient algorithm for performing systematic search is *backtracking*. It incrementally attempts to extend a partial assignment that specifies consistent values for some of the variables, toward a complete assignment, by repeatedly choosing a value for another variable consistent with the values in the current partial solution. If a partial assignment violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has available alternatives. Whenever a partial assignment violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. Consequently, backtracking is strictly better than generate-and-test.

The late detection of inconsistency is the disadvantage of GT and backtracking paradigms. Therefore, various consistency techniques for constraint graphs were introduced to prune the search space. They try to eliminate values that are inconsistent with some constraints. Thus, the inconsistency is detected as soon as possible. The consistency techniques range from simple *node consistency* and the very popular *arc consistency* to full, but expensive

path consistency.

Neither systematic search nor consistency techniques prove themselves to be efficient enough to solve the CSP completely. Various schemes that combine these two were introduced. They integrate a consistency algorithm inside a search algorithm. They are based on the idea of reducing the search space through constraint propagation. Such inference (i.e. constraint propagation) is useful since it may reduce the parts of the search space that need to be visited.

4 The bytecode sequence construction problem as a CSP

As explained in section 2.3, the code sequence construction problem could be presented as follows: given a start state and an arrival state we have to find instructions sequence to add, respecting a set of defined constraints. This goes back to a constraints satisfaction problem. The construction of that sequence must solve two main problems: choosing an instruction among the existing ones and computing the memory state preceding it in order to reach the desired state.

According to our researches, no similar problem has been discussed before in the literature. However, we were inspired by the works of Charreteur and Gotlieb presented in [17]. They introduced a constraint-based reasoning approach to automatically generate test input for Java bytecode programs. Their memory model is based on the notion of *constrained memory variables* (CMV) which captures Java Virtual Machine states. Each Java bytecode will then be seen as a relation among two CMVs: the CMV before activation of bytecode and the CMV after its activation. An innovative aspect of their approach is the definition of a constraint model for each bytecode that allows backward exploration of the bytecode program. It is precisely this capacity that interests us in order to calculate the memory state preceding a chosen instruction during the construction of our sequence.

To formalize a problem as a CSP, we must identify a set of variables, a set of domains and a set of constraints. Our bytecode sequence construction problem is defined as follows:

Given:

- A finite set of variables X, representing all the bytecode instructions to add (the sequence to be found)
- A discrete finite domain D, representing the set of the bytecode instructions defined in the specification [46]. Each variable in X takes a value from D i.e. the values of the instantiated variables will be selected from this set.
- A set of constraints C representing all the constraints to be satisfied while the sequence construction (instantiation of the variables of X).

Find:
A set of solutions, where a solution is a sequence of bytecode instructions such that each instruction is an instantiation of a variable x_i from X respecting a set of constraints.

4.1 Variables and domains

We choose to make each instruction to find as a variable to be instantiated. The set of variables is $X = \{x_1, x_2, ..., x_n\}$ where n is the length of our sequence to find. A bytecode instruction is defined by its opcode and zero or many operands. Thus, instantiate a variable x_i consist to associate an opcode (the unique identifier of a bytecode instruction).
Each one of these variables can take a value from the same domain D which contains all the possible opcodes as defined in the specification [46]. The D domain is a finite set of hexadecimal values from `0x00` to `0xB8`. Thus, D ={`0x00, 0x01, 0x02, ..., 0xB7, 0xB8`}.

4.2 Constraints

In our case, we define two categories of constraints:

General constraints common to all the instructions i.e. the instantiation of each variable x_i must respect them. We can list the following constraints:

- The operand stack size must not exceed the value stored in the header of the method (called *maxStack*),
- The number of local parameters is fixed (called *maxLoc*),
- The chosen instruction must not cause an overflow or underflow in the operand stack,
- The produced/consumed elements for the chosen instructions using local variable must be compatible with the local variable list,
- The types of produced elements for each chosen instruction must be compatible with the current operand stack state, *etc.*

Specific constraints proper to each bytecode instruction. They are based on the semantics of each instruction as defined in the Oracle specification [46]. This set of constraints could change dynamically depending on the value that one variable takes. Each instruction has three constraints. They concern:

- Pre-condition, the types of elements being consumed by the instruction.
- Post-condition, the types of elements being produced by the instruction.
- The use or not of local parameters and the corresponding couple (type, index) if yes.

Below, we present two examples of bytecode instructions and their corresponding specific constraints.

Example1:
As indicated below (from [46]), the instruction `sadd` (its opcode is `0x41`) has zero operand, it pops (consumes) two short values from the top of the operand stack, sums them up and then put the result, which is also a short value, on the top of the stack. It does not use local variables.

sadd
Add short

Format
sadd

Forms
sadd = 65 (0x41)

Stack
..., value1, value2 –>
... , result

Description
Both value1 and value2 must be of type short. The values are popped from the operand stack. The short result is value1 + value2. The result is pushed onto the operand stack.
If a sadd instruction overflows, then the result is the low-order bits of the true mathematical result in a sufficiently wide twos-complement format. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

So the constraints associated to this instruction are:

- $Pre - condition = \{short, short\}$
- $Post - condition = \{short\}$
- $Locals = \{\phi\}$

Example2:
As indicated below (from [46]), the instruction `aload2` (its opcode is `0x1a`) has zero operand, it does not need any element (empty pre-condition) and pushes an objectref element from the local variable at index 2 onto the top of the stack. The local variable at that index must contain a reference.

$aload < n >$
Load reference from local variable

Format
$aload < n >$

Forms
aload0 = 24 (0x18)
aload1 = 25 (0x19)
aload2 = 26 (0x1a)
aload3 = 27 (0x1b)

Stack
... –>
... , objectref

Description
The $< n >$ must be a valid index into the local variables of the current frame. The local variable at $< n >$ must contain a reference. The objectref in the local variable at $< n >$ is pushed onto the operand stack.

So the constraints associated to this instruction are:

- $Pre - condition = \{\phi\}$
- $Post - condition = \{objectref\}$
- $Locals = \{(objectref, 2)\}$

5 Elements of modelization

Starting from our problem formulation as a CSP (section 4), we aim to solve it i.e. find assignments to its variables while respecting all the defined constraints. The search for a solution to a CSP may be viewed as tree traversal. An important aspect of the search considered here, is that the tree to be traversed is not given in advance: it is generated on the fly.

5.1 General structure

Given a starting memory state (representing the memory state before the execution of the first instruction of the code to be hidden), a final memory state and the list of all bytecode instructions (among which we will choose the instructions to be added), we may build a search tree where:

- The *root* represents the starting memory state

– Each *level* contains the candidate instructions (i.e. those respecting the constraints) which can be part of the sequence to obtain in the sense that they can precede the instruction of the higher level.
– Each *node* represents a memory state corresponding to a candidate instruction i.e. an instruction preceding the parent node (we remind that we reason in the opposite direction of the execution).
– Each *leaf* represents a final memory state (the desired state).

In the figure below (figure 3), each choice is an abstraction of a memory state.

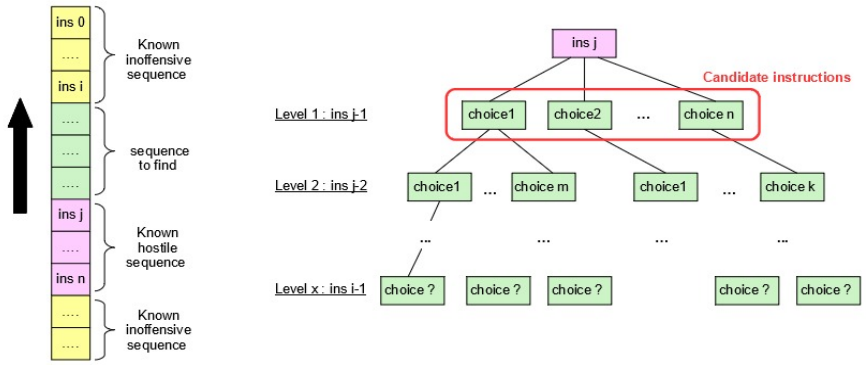

Fig. 3: General structure

5.2 Node modeling

The structure of each node of the search tree is divided into two parts (figure 4):

1. *Data*, to manage the constraints during the search tree generation, it includes:
   – The current operand stack
   – The current local variables list
   – The stack pointer indicating the top of the operand stack
   – The candidate instruction (its opcode)
2. *Pointers*, to ensure the tree traversal (in both the ascending and descending orders), it includes:
   – A pointer to the parent node
   – A list of pointers to the potential sons (each one corresponds to a candidate instruction)

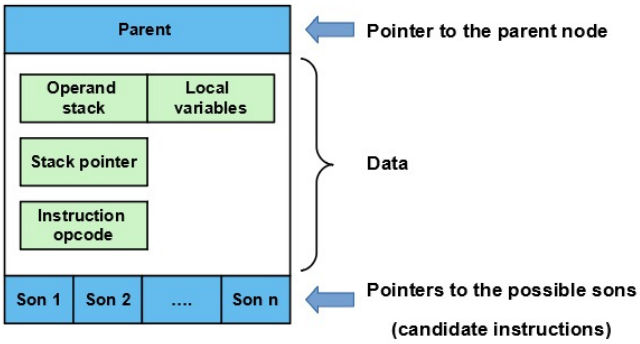

Fig. 4: Node structure

6 Proposed approach

In this section, we first present our tree traversal algorithm (section 6.1), and then we detail how to calculate a memory state (section 6.2). After that, we illustrate the proposed code sequence construction approach on an example (section 6.3). Finally, we present the next step after the generation process: CAP[9] file manipulation and verification (section 6.4).

6.1 Tree traversal Algorithm

As explained in section 3, many techniques exist to generate and explore a search tree. The algorithm 1 represents our tree traversal approach to generate bytecode sequences which relies two memory states. It is based on a *depth-first strategy* and two stop criteria:

- Reach a desired final state (a leaf node)
- Reach the maximal depth (its value is fixed in advance)

The objective of our approach is therefore, from a starting memory state, to insert instructions and to recalculate the previous memory state in order to converge to the arrival memory state to join. Each path from the root to a leaf (in the ascending direction of execution) represents a possible solution i.e. a possible bytecode sequence to be added to the program. Thus, the construction of that sequence must solve two main problems: the choice of the instructions while respecting the constraints (this is ensured by constraints propagation before each choice) and the calculation of the corresponding memory states. The decision function of the instructions choice must be accompanied by a *backtracking* mechanism if a stop criterion is encountered.

[9] CAP file (Converted Applet) is a converted Class file adapted for the resource-limited devices

Algorithm 1: The construction of code sequence by tree traversal

input : A starting memory state + an arrival memory state + A set of bytecode instructions
output: A bytecode sequence joining the two memory states

```
1  begin
2      Starting from the tree root, find all the candidate instructions (those
         respecting the defined constraints i.e. can preceed the root);
3      while stop criterion is not reached do
4          Generate all the son nodes that correspond to the found candidate
             instructions (one node by instruction);
5          Select a non visited son node to explore;
6          Compute the memory state of the selected node and mark it as visited;
7      end
8      if the maximal depth is reached then
9          ** if the desired memory state is reached then
10             Memorize the solution (the path from the root to the current node);
11             * if a parent node still have a non visited son then
12                 backtrack to this node and take it as the current node;
13                 go back to line 3;
14             else
15                 Terminate the search (the root have no more son to explore);
16             end
17         else
18             Go to *
19         end
20     else if the desired state is reached then
21         Go to **
22     end
23 end
```

Starting from the root of the tree, at each level of the tree we generate the possible sons (intermediate nodes representing the candidate instructions that respect the general and specific defined constraints) and proceeds by descending to the first son. This process continues as long as a node is not a leaf and the maximal depth is not reached. If a stop criterion is encountered, the search proceeds by moving back to the parent of the current node. Then the next non visited son, if any, of this parent node is selected. This process continues until the control is back to the root node and all of its sons have been visited. All the paths from the root to the leaves represent the possible solutions to the code construction problem.

We need to generate several solutions (although theoretically only one is sufficient to validate the approach) because the found solutions must undergo a verification phase in order to determine which ones are valid in the sense that they can be accepted by the BCV (see section 6.4). So by generating several solutions we have a greater probability of finding a good solution.

6.2 Memory state computation

The memory state computation includes:

- The calculation of the operand stack state
- The update of stack pointer
- The update of the local variables list

As explained before, we are working backwards. So, for each candidate instruction (son node), instead of removing its consumption from the stack and adding its production onto the stack, we have to pop the produced elements (post-condition) and push the needed elements (pre-condition).

Thus, for each node creation, the stack state can be obtained as follows:

`(Son_Stack_State)` = `(Father_Stack_State)` - `(Post-condition of the candidate instruction)` + `(Pre-condition of the candidate instruction)`

6.3 An example

To illustrate the proposed approach, we present an example of the execution of algorithm 1 (figure 5). To simplify the representation of the tree we chose to restrict each memory state to its operand stack state and current bytecode instruction.
In order to have a search tree of reasonable size to present in this paper, we have selected a subset of bytecode instructions to consider when generating/exploring the tree. Table 3 summarizes the chosen instructions as well as their *specific constraints* (pre-condition, post-condition and the use of local variables).

Instruction	**Pre-condition**	**Post-condition**	**Local variables**
`aload_0`	none	ObjectRef	(ObjectRef, 0)
`aload_1`	none	ObjectRef	(ObjectRef, 1)
`bspush`	none	short	none
`getfield_a_this`	none	ObjectRef	none
`sadd`	short short	short	none
`sconst_2`	none	short	none
`sload_0`	none	short	(short, 0)
`sload_1`	none	short	(short, 1)

Table 3: pre/post-conditions for a subset of bytecode instructions

Consider the following inputs :

- Initial instruction = `sload_1`
- Initial operand stack state = $\{short, ObjectRef, Short\}$
- Final operand stack=$\{\phi\}$
- Local variables = $\{ObjectRef, Short\}$

– MaxStack = 4
– Maximal depth = 3

Fig. 5: example of execution

Starting from the `sload_1` instruction (root node), we browse the instructions list (table 3) to select the candidate instructions corresponding to the current memory state. In other words, we look for those instructions whose post-condition matches with the top (one or more elements) of the current operand stack and local variables states. Prior to that, all the *general constraints* may be respected (see section 4.2). Thus, the `sload_1` can be preceded by `bspush, sadd, sconst_2` or `sload_1` (`sload_0` is incompatible because the local variable at index 0 is a reference and not a short).
It is worth to recall that we perform a depth-first search while generating/ exploring our tree. As no priority is established between the sons, i.e. candidate instructions, we take the first one and repeat the process. So, the `bspush` instruction can be preceded by `aload_0` or `getfield_a_this`. After that, we se-

lect the candidate instructions for `aload_0` which are `bspush, sadd, sconst_2` or `sload_1`. In the next step when selecting the `bspush` instruction we find that the maximal depth is reached and the current operand stack state corresponds to the desired one (empty stack). Consequently, the path from the root to this instruction is saved as a possible solution. Then, we backtrack to the parent node (`aload_0`) and explore the next son (`sadd`). The maximal depth is reached and the operand stack state is not the desired one. So, we backtrack to the parent node and repeat the same process until the root node has no more sons to visit.

6.4 Next step : CAP file manipulation and verification

The list of solutions previously found may be analyzed to detect which of them are valid i.e. could be generated by a compiler. We proceed by two steps as presented in figure 6.

The first step consists of *CAP file manipulation*. We aim to insert each found solution, i.e. a bytecode sequence, into a correct CAP file (using the *Cap manipulator tool*[10]) and only keep those that can be accepted by the byte code verifier (BCV). We produce as many CAP files as solutions since every CAP file is the original one plus the added sequence. At this step, two main problems are resolved: adapt the solution's format to the CAP file one (as defined in the specification [46]) and give the correct arguments to some instructions (e.g. jump offset, referenced argument).

In the second step, we perform a *CAP file verification*. Each Cap file created in step one, is converted to a class file and then to a Java file. With this Java file we create a Class file then a CAP file. At the end we compare the original CAP file (created in step one) and the one just created. If the two files are identical, we can conclude that the corresponding program could be generated by a compiler (an accepted solution).

7 Approach Optimisation: Heuristics

In the proposed approach (section 6), the candidate instructions (sons) were explored with no preferences i.e. with the same chance, which is not really efficient in practice. In other words, combinations that do not correspond to real programs will be explored. A way to optimize the search process is to introduce heuristics for a faster convergence towards more realistic solutions. With heuristics, instructions will be weighted so that they will be ordered and explored according to their priorities.

[10] Available on: https://bitbucket.org/ssd/capmap-free

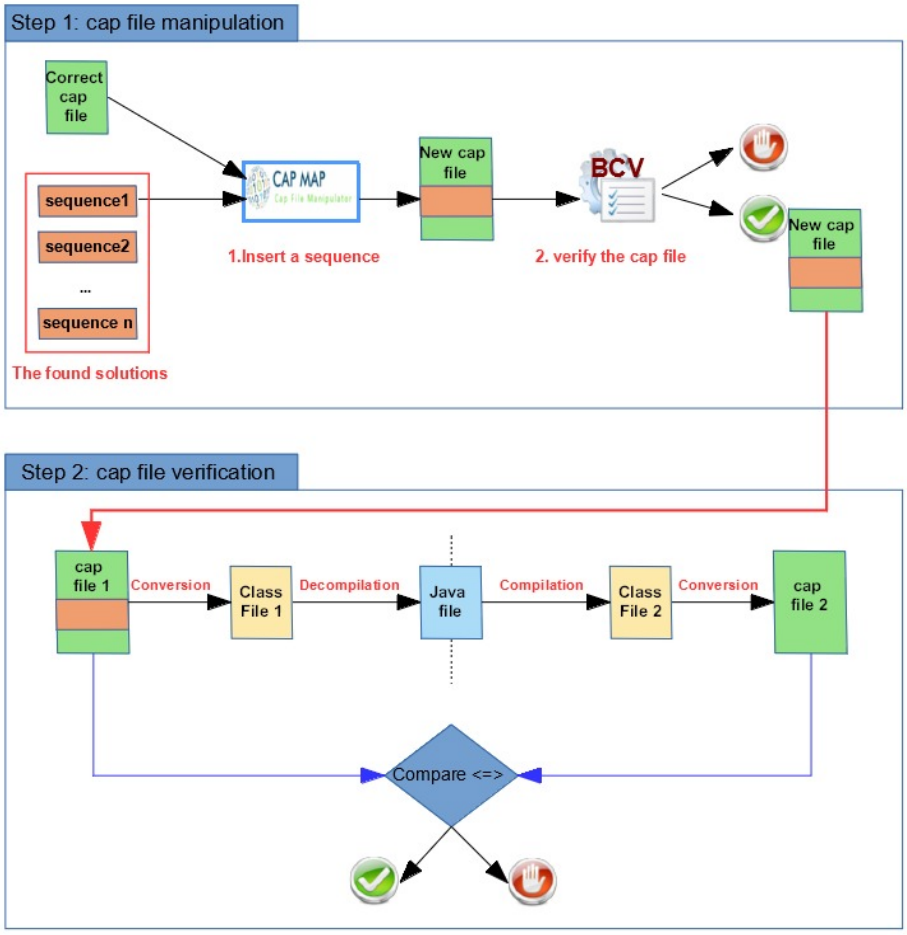

Fig. 6: CAP file manipulation and verification process

The first step of this approach is the generation of statistics files for each instruction, based on the frequency of transition to the following instructions recorded from a set of Java Card applets. Thus, when a generation step is initiated, the statistics corresponding to the current instruction are loaded. These statistics include the instruction names sorted by frequency of transition (the first instruction is the most frequent).

We developed two types of statistics (depending of the nature of the considered node):

- Bigram statistics, applied for the root node of the search tree
- Trigram statistics, applied for the other nodes (intermediate) of the tree

7.1 Bigram statistics

A *Python* script was developed to analyze *.jca*[11] files to extract the bytecode and calculate the statistics of transition for each instruction. First, the script generates a transition matrix between all the instructions, it corresponds to the frequency of each precedent instruction. With this matrix the script produces, for each instruction, a table with a sorted list of instructions from the most used to the least used. An example of the generated bigram statistics for some instructions is given in table 4.

Instruction	**Possible previous instruction**
`aaload`	`sload_1 sload sload_3 sload_2 ssub getfield_b_this sconst_0 ...`
`aconst_null`	`aload_0 areturn ifnull new putfield_a putstatic_a`
`astore`	`goto checkcast aaload newarray aload_1 getfield_a_this aload_3 ...`
`bspush`	`bspush dup sconst_0 aload_1 aload_0 sload baload aload_2 ...`
`if_acmpeq`	`aaload aload`
`sadd`	`sconst_1 sload sconst_2 bspush sadd sconst_4 sload_2 sload_3 ...`

Table 4: Example of bigram statistics

With this solution, the statistics file gives different possible previous instructions for each instruction. Moreover, the number of sons is different from one instruction to another. This reduces the number of possible branches for a node and allows for a faster convergence to more realistic solutions.

7.2 Trigram statistics

Another *Python* script was developed for a modified exploitation of the *.jca* files. Trigram statistics files are computed to direct the sons selection for each node based on the knowledge of the parent instruction. A table of possible sons is generated for each instruction (each table is stored in an independent file). According to figure 7, we suppose that the current instruction is *ins2*. We try to find the *ins1* (all possibilities of son nodes) such that *ins2* is preceded by *ins1* and followed by *ins3* (parent node).

Each trigram statistics file is organized as follows:

- The name of the file is the current instruction name.
- The first column is the next instruction (the parent node in our search tree).
- The rest of the row is the possible previous instructions (son nodes i.e. the candidate instructions).

Thus, when the sons generation is performed, the program opens the statistics file which corresponds to the name of the current instruction. It searches the line in the first column, which corresponds to the name of its parent node, and loads the rest of the row as the candidate instructions to explore.

[11] A JCA (Java Card Assembly) file is a text representation of the contents of a CAP file

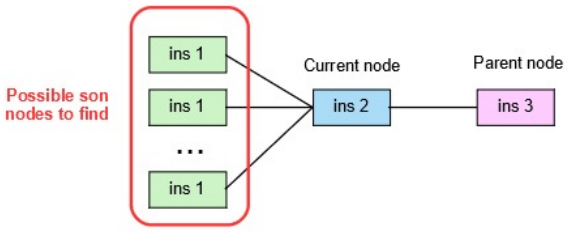

Fig. 7: Trigram concept

sload	
Next instruction	**Possible previous instructions**
`aaload`	`getfield_a_this`
`aload`	`aload sload sadd sstore`
`baload`	`getfield_a_this aload_2 aload_1 aload_3 aload aload_0`
`dup`	`aload_2 aload_3 getfield_a_this aload`
`sload`	`getfield_a_this sstore sinc aload_2 aload_0 astore sload`

Table 5: Example of trigram statistics for *sload* instruction

As shown in table 5, the instruction `sload` which is followed by an `aload` (parent node) could be preceded by (possible son nodes): `aload`, `sload`, `sadd`, `sstore` in the descending order of priority.

8 Trace Generator tool

The proposed approach is implemented through a tool named *Trace Generator* (developed in Java). Its architecture is represented in figure 8. It takes several inputs, performs the search tree generation/exploration (the optimized version of the approach, see section 7) and restitutes a text file containing all the found solutions. The tool provides two possible generation modes: *classic* or *random*.

- *Classic mode*: generates all possible solutions with a defined depth in the tree. The selection of son nodes to explore is based on statistics (section 7). We need to visit all possible sons of the current node before exploring another one at the same level (depth-first search).
- *Random mode*: This mode is still based on the trigram statistics to find the candidate instructions (sons), but the selection of each initial son is randomly made. Furthermore, the tree generation is restarted to the root after a certain number of found solutions. This allows to increase the solutions diversity i.e. the produced successive solutions are really different from each other. Its useful if we choose to generate a reduced number of solutions with a high diversity, even with great depth in the tree.

To resume, according to our need, we can use the classic mode to generate all the solutions selecting the best statistics or the random mode to generate

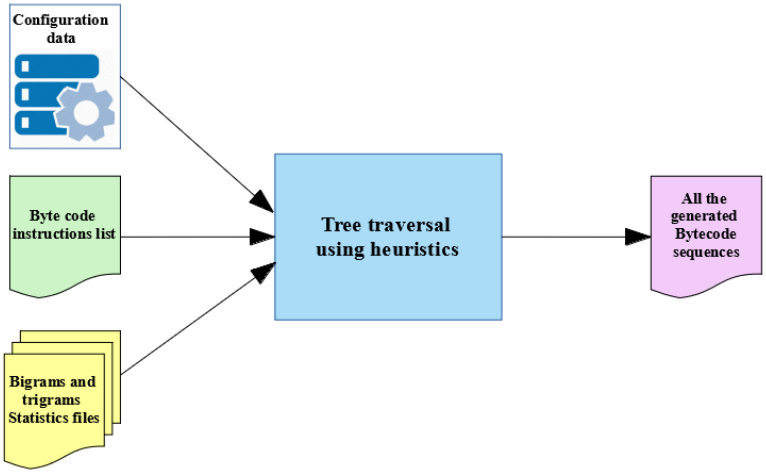

Fig. 8: Trace Generator tool architecture

a partial set of solutions but with a higher diversity between two successive ones.

The tool needs three main inputs:

1. The bytecode instruction list represented in a special format (figure 9) to facilitate the extraction of the specific constraints (section 4.2) during the tree traversal.
2. The obtained bigrams and trigrams statistics files.
3. The configuration data including:
   - The generation mode (classic or random)
   - The start instruction (root node)
   - The maximal depth i.e. number of levels in the tree (corresponding to the maximal solution length)
   - The maximal number of solutions
   - The initial memory state (resp. the arrival memory state) represented by the operand stack state and initial local variables list
   - The number of solutions before root return if the chosen mode is random

The tool's output is a text file including all the found solutions i.e. all the paths from the root to leaves. These solutions will be subject to a verification phase, as explained in section 6.4, to decide if they are accepted or not. Note that the current version of the tool handles a meaningful subset of Java Card bytecodes instructions.

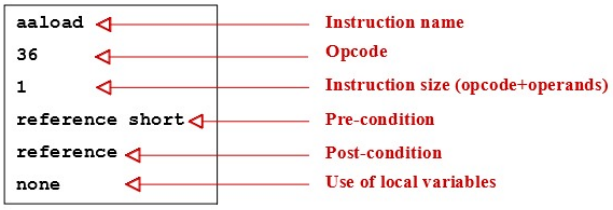

Fig. 9: Example of the representation of a bytecode instruction in the input file

9 Experimental results

The aim of this section is twofold: we want to show the utility and the efficiency of our approach of code sequence construction (through the implemented tool) while giving a proof-of-concept of the desynchronization mechanism.
In order to perform this, we choose a practical case based on the results presented by Mesbah et al. in [41]. In their paper, they presented a new approach for reversing the unknown instruction set of the intermediate bytecode which in turn has led to reverse engineering of the Java classes of the attacked card. They discovered during the reverse that some method calls have an unusual signature. Without having access to the native code, the semantics of the called methods and their calling convention have been inferred. These methods have access to the assets of the card without being restricted by security mechanisms like the firewall. This knowledge was exploited to set up a new attack that provides a full access to the cryptographic material and allows to reset the state of the card to its initial conguration. The authors demonstrated the ability to call these methods at the Java level in an application to retrieve sensitive assets whatever the protections are.
Our goal is to hide native calls found in [41] based on the proposed approach in section 6. The card uses some specific headers to represent native methods. Moreover, it uses a specific instruction to call them. It is the instruction `0xCD` which needs two bytes used as token for the native call. The found native calls [41] are resumed in table 6.

As stated before, the main focus in this paper is not about the desynchronization problem. Therefore, we will consider a basic case for our proof-of-concept. The other cases will be treated more in details in future work.
The goal is to hide the call of a native method, we try to insert just before this call an instruction, its opcode only, which takes 2 operands (that correspond to `0xCD op1`). This will cause the desynchronization of the original code. In other words, as presented in figure 10, the two bytes (`0xCD op1`) will be the operands of the added instruction and the second operand `op2` will be interpreted as a new instruction. However, in addition to the choice of the instruction to be added (*BF* for *Brute Force*, in what follows), the preceding

Native call (0x)	Method's name	Native call (0x)	Method's name
CD 21 80	readByteVMSTACK()	CD 22 42	writeByte()
CD 21 80	writeByteVMSTACK()	CD 33 42	writeShort()
CD 21 C0	readShortVMSTACK()	CD 73 42	xorify()
CD 22 C0	writeShortVMSTACK()	CD 62 42	deXorify()
CD 21 00	readByteRam()	CD B1 0F	deadCard()
CD 22 00	writeByteRam()	CD A3 1C	generateRandomData()
CD 21 40	readShortRam()	CD A1 1D	isAppletActive()
CD 22 40	writeShortRam()	CD 25 AB	encryption()
CD 22 02	readByte()	CD 25 2B	decryption()
CD 33 02	readShort()		

Table 6: Native Methods

code sequence (preamble) must be generated in order to have the required memory state for the correct execution of the code. Our proposed approach of code sequence generation will be applied at this stage. The final result will be a new piece of code where the native call has been hidden while keeping the code syntactically and semantically correct (constraint satisfaction).

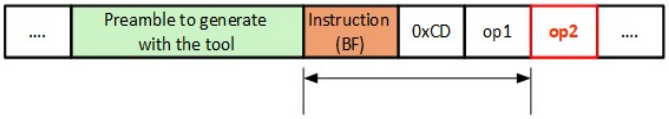

Fig. 10: Hiding a native call

Given the set of native calls and the set of bytecode instructions with two operands, we studied the possible cases to consider. We proceeded by elimination to reduce the sets to be treated in the experiments. The followed steps are:

1. The choice of the native calls to consider
2. The choice of the instruction BF
3. The generation of the sequence to be added (use of the *trace generator* tool)

9.1 Step 1: Choosing the native call to consider

Since the first two bytes of a native call will represent the operands of the BF instruction (`0xCD op1`), the remaining byte (`op2`) will be interpreted as the opcode of another bytecode instruction. The table below (table 7) lists the possible values of this byte (the corresponding instructions) as well as the pre-condition for each one.

According to table 7, we have four possible cases:

Case	op2 value	Instruction	Pre-condition	Native call (0x)
Case 1	0xC0	Reserved instruction	?	CD 21 C0
				CD 22 C0
Case 2	0x00	`nop`	none	CD 21 00
				CD 22 00
	0x02	`sconst_m1`	none	CD 22 02
				CD 33 02
	0x0F	`iconst_5`	none	CD B1 0F
	0x1C	`sload_0`	none	CD A3 1C
	0x1D	`sload_1`	none	CD A1 1D
Case 3	0x80	`putstatic_b`	byte	CD 21 80
				CD 22 80
	0xAB	`getfield_s_w`	ObjectRef	CD 25 AB
	0x2B	`astore_0`	ObjectRef	CD 25 2B
Case 4	0x40	`swap_x`	many bytes	CD 21 40
				CD 22 40
	0x42	`iadd`	int int	CD 22 42
				CD 33 42
				CD 73 42
				CD 62 42

Table 7: op2 values and corresponding instructions

- *Case 1:* the value of `op2` is `0xC0` which correponds to a reserved bytecode instuction according to [46]. Thus, the corresponding native calls are not considered.
- *Case 2:* the value of `op2` corresponds to an instruction whose pre-condition is empty. Such instruction is not affected by the post-condition of the *BF* instruction. So, it can be executed without problem.
- *Case 3:* the value of `op2` correponds to an instruction whose pre-condition is equal to the *BF* instruction's post-condition. It can be executed without problem, too.
- *Case 4:* the value of `op2` corresponds to an instruction whose pre-condition requires one or many elements that are not compatible with the *BF* instructions post-condition. Such instructions need more inversigation and are not considered in the scoop of this paper.

9.2 Step 2: Choosing the BF instruction

The Oracle's virtual machine specification [46] defines 43 bytecode instructions having two operands. After studying the possible cases for the *BF* instruction (according to the partial instruction set considered by our tool and realistic values of the different operands), we retained the 10 following instructions: `anewarray, getstatic_a`, `getstatic_b, getstatic_s`, `jsr, new`, `putstatic_a`, `putstatic_b`, `putstatic_s, sspush`. The table below (table 8) lists the considered instructions as well as their pre-conditions and post-conditions.

Instruction BF	Opcode (0x)	Pre-condition	Post-condition
`anewarray`	91	short	arrayref
`getstatic_<t>`	7B .. 7E	/	short, ObjectRef
`jsr`	71	/	address
`new`	8F	/	objectref
`putstatic_<t>`	7F..82	`value_<t>`	/
`sspush`	11	/	short

Table 8: BF instruction possibilities

9.3 Step3: Generation of the sequence to add

In this generation step, for each considered native call (step1), we take all possible cases of the *BF* instruction (step2) and generate the possible solutions using the *Trace Generator* tool. Depending on the maximum size of the sequence (it represents the search tree's depth) and the maximum number of solutions (which ranges from 100 to 1 million), the number of the found solutions and the required generation time are recorded.

To perform this, we considered the following tool inputs:

- Starting instruction: the instruction BF (10 choices)
- Starting stack memory state: pre-condition of the considered *BF* instruction
- Arrival stack memory state: empty stack
- List of local variables e.g. `reference, short, reference`
- Generation mode: classic

We notice that the instruction *BF* does not depend on the value of the bytes of the native call (its 2 operands), whatever the considered native call is, we will have the same experimentation. So, the generation results are resumed in one table regardless of the considered native call. The experiments were done on an i7-6500u 3.1Ghz processor PC. The obtained results are summarized in table 9.

Through these results we notice that we have two categories of instructions:

- For the instructions presented in table 9 (`getstatic_a, getstatic_s, putstatic_a`, `sspush`), we can have a number of solutions which exceeds 1000 in a couple of seconds and more than 1 million in a couple of minutes (between 7 and 14.5 minutes). Which is a very reasonable time for that amount of different solutions. The found solutions are subject to a verification step (section 6.4) to be loaded in a real card.
- Instructions not presented in the table above have a single solution regardless of the maximum size of the solution. This is due to:
  - The fact that in the bigrams statistics file, the corresponding line is empty (i.e. no data). That is to say, the root node (the considered BF instructions) does not have candidate instructions to precede it. For example: `putstatic_b, putstatic_s, jsr`. This can be improved by enriching the statistics file.

solution	*getstatic* − *a*		*getstatic* − *s*		*putstatic* − *a*		*sspush*	
size	# sol	time ms	# sol	time ms	# sol	time ms	# sol	time ms
1	3	4	2	3	1	3	4	3
2	5	6	4	4	2	3	20	9
3	23	8	15	6	2	3	91	26
4	47	24	31	14	3	10	289	88
5	178	55	99	33	16	15	1045	294
6	581	200	286	126	27	53	3423	1123
7	2123	743	1140	403	98	210	13378	4489
8	7663	3038	3746	1698	516	350	49313	17909
9	31833	11886	16001	6880	1651	1393	200379	71205
10	117080	48117	54889	27070	6275	5736	733922	288127
11	464974	219854	231265	109069	25148	23613	>1000000	410241
12	>1000000	510000	844996	511202	108213	105199		
13			>1000000	860946	425547	440450		
14					>1000000	448738		

Table 9: Experimental results obtainted by the Trace generator tool

– The *BF* candidate instructions do not find their post-conditions on the stack, i.e. the post-condition is different from the pre-condition of the *BF* instruction. For example: `new, getstatic_b`. This case could be studied in order to have more solutions. This is possible by finding the appropriate initial memory state (compatible with the post-conditions of the candidate instructions). But the effect on the codes postamble (i.e. code which comes after the native call) remains to be verified, too.

9.4 An example

In order to illustrate the obtained results, we present two examples of hiding a native call based on the explanations given in the above sections. However, other examples of possible combinations (native call, BF instruction) can be realized by following the same steps. Prior to that, according to tables 7 and 8, the considered native calls (case 2 and 3 of section 9.1) and their corresponding BF instructions are summarized in table 10.

9.4.1 Example1: op2 instructions pre-condition is empty

This example belongs to case 2 (section 9.1). We try to hide the call to the native method `readByteRam()` (i.e. it is considered as the hostile code to dissimulate). For that, we chose the instruction `putstatic_a` as a BF instruction to add before the native call. The resulting code is presented in figure 11. The `putstatic_a` instruction takes the two first bytes of the native call as its operands (i.e. `0xCD 0x21`). Thus, the remaining byte is interpreted as a `nop` instruction (the value of `op2` was `0x00`).

It remains to generate the preamble sequence matching with the BF instructions pre-condition. The *Trace Generator* tool provides us with a set of possible

op2 value	Instruction	Pre-condition	Native call (0x)	BF instruction
0x00	`nop`	none	CD 21 00	All the BF instructions considered in section 9.2
0x02	`sconst_m1`	none	CD 22 00 CD 22 02 CD 33 02	
0x0F	`iconst_5`	none	CD B1 0F	
0x1C	`sload_0`	none	CD A3 1C	
0x1D	`sload_1`	none	CD A1 1D	
0x80	`putstatic_b`	byte	CD 21 80 CD 22 80	`getstatic_b`, `getstatic_s`, `sspush`
0xAB	`getfield_s_w`	ObjectRef	CD 25 AB	`new, getstatic_a`
0x2B	`astore_0`	ObjectRef	CD 25 2B	`new, getstatic_a`

Table 10: The considered native calls and their corresponding BF instructions

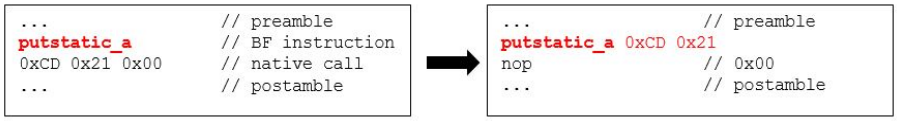

Fig. 11: Hiding the native call to method readByteRam()

solutions. Among which we present the following one (figure 12) accompanied by its execution (stack states).

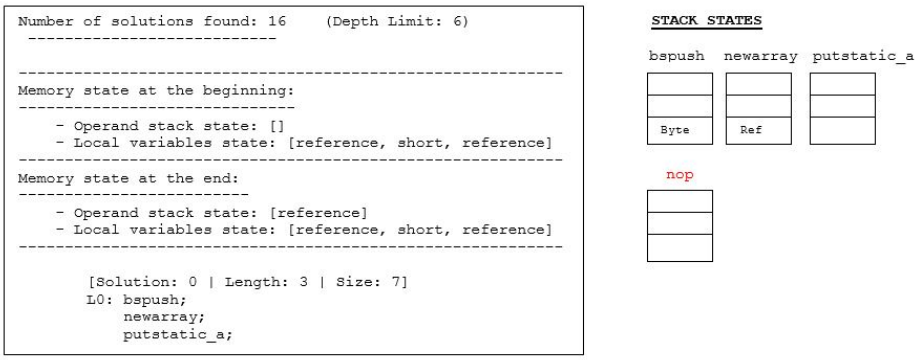

Fig. 12: An example of a solution generated by the tool (1)

9.4.2 Example 2: op2 instructions with a pre-condition

This example belongs to case 3 (section 9.1). We recall that in this case the `op2` instruction's pre-condition corresponds to the BF instructions post-condition. We want to hide the call to the native method `decryption()`. For that, we

chose the instruction `getstatic_a` as the BF instruction. The resulting code is presented in figure 13. The `putstatic_a` instruction takes the two first bytes of the native call as its operands (i.e. `0xCD 0x25`). Thus, the remaining byte is interpreted as an `astore_0` instruction (the value of `op2` was `0x2B`).

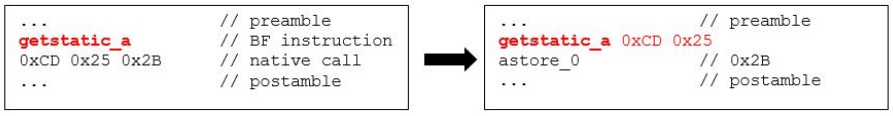

Fig. 13: Hiding the native call to method decryption()

Among the different possible solutions generated by our tool, we present the following one (figure 14) accompanied by its execution (stack states).

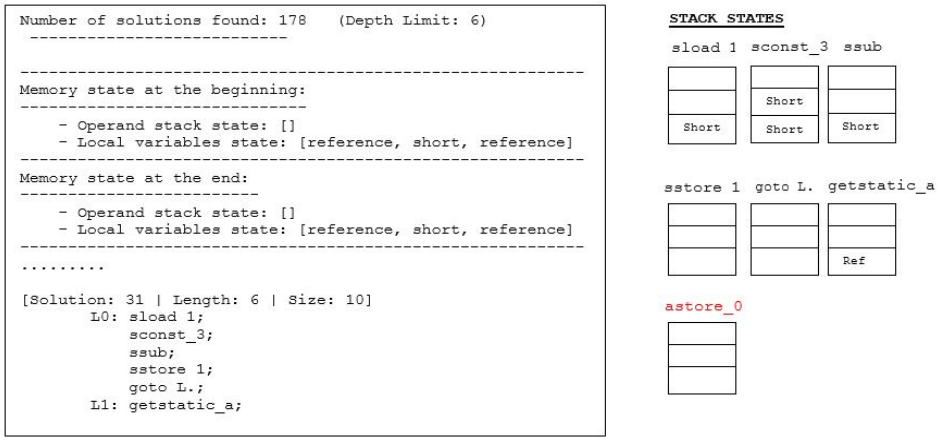

Fig. 14: An example of a solution generated by the tool (2)

Regardless the treated case, if after the fault injection there are still elements on the stack (generated by the preamble), they must be managed in the postamble (to be consumed). This is performed in order to recover the original flow of the execution.

10 Conclusion and future work

In this paper, we have presented a new risk related to a particular fault attack on a legitimate application which mutates intentionally in a hostile way. Designing an applet for this purpose still quite difficult but we showed that it is possible. Based on a theorical foundation, we have shown that this problem

comes back to a constraint satisfaction problem. Indeed, we try to dissimulate a hostile code into an inoffensive one by adding some instructions: it corresponds to code construction. The added code is a sequence of instructions chosen from those defined in the specification while respecting a set of constraints. This is to ensure that the resulting code is syntactically and semantically correct with respect to the specification. The construction of that sequence must solve two problems: choosing an instruction among the existing ones and computing the memory state preceding it in order to reach the desired state (an inoffensive piece of code).

As a solution to this problem, we presented a novel approach of backward code construction based on constraints satisfaction and a tree traversal algorithm. The idea is to represent this problem as a search tree in which the root is the first instruction of the hostile code and at each level the nodes represent the candidate instructions that may precede the parent one (we recall that we reason in backward direction). The tree traversal algorithm is used to create and explore the tree in order to find paths from the root to the leaves (representing the desired state). Each one of these paths corresponds to a possible wanted sequence. To improve the quality of the generated solutions, we proposed an optimisation based on heuristics (bi-grams and tri-grams) in order to converge towards more realistic solutions in a faster way. The idea is to weight instructions so that they will be ordered and explored according to their priorities. To perform this, statistical files were created based on the frequency of transition to the following instructions recorded from a set of selected Java Card applets.

The proposed approach has been put into practice through the implementation of the *Trace Generator* tool. It takes as input the initial and arrival memory states (in addition to other configuration parameters) to automatically generate the possible sequences linking them. In order to show the efficiency of the tool, a proof-of-concept of the desynchronization principle has been realized. The considered case study is to hide a native methods call into a code that has been built and that once activated through a fault injection we will have the desired hostile behaviour. As the main focus of this paper is not about desynchronization, we limited the case study to a basic case of this latter. The obtained results confirm that we can effectively apply our approach to a practical case that aims to hide hostile behaviour which could have dangerous consequences threatening the security of the card. The proposed approach could be applied to various hostile codes regardless of the intention of the attacker (i.e. it aims to conceal the hostile code regardless of its content). Moreover, although our approach has been applied to smart cards, other secure elements should also be an interesting target. In this case, it will be enough to adapt the considered language in the model (CSP part) to the specificities of the target element without any change in the general reasoning.

As we have already mentioned, the problem of desynchronization will be the subject of our next step in order to consolidate the present work. Indeed,

we aim to have a more complete formalization of the principle by treating in more depth all the cases that may arise in order to draw generic conclusions about desynchronization as a new technique of code dissimulation. Once this is done, we will tackle countermeasures. In other words, how to protect sensitive assets against a fault enabled virus?

References

1. Douglas B Armstrong. A deductive method for simulating faults in logic circuits. *IEEE Transactions on Computers*, 100(5):464–471, 1972.
2. Mark W Bailey, Clark L Coleman, and Jack W Davidson. Defense against the dark arts. *ACM SIGCSE Bulletin*, 40(1):315–319, 2008.
3. Arini Balakrishnan and Chloe Schulze. Code obfuscation literature survey, 2005.
4. Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.
5. Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java Card operand stack: fault attacks, combined attacks and countermeasures. In *International Conference on Smart Card Research and Advanced Applications*, pages 297–313. Springer, 2011.
6. Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on Java Card 3.0 combining fault and logical attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 148–163. Springer, 2010.
7. Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. Low voltage fault attacks on the RSA cryptosystem. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 23–31. IEEE, 2009.
8. Elena Gabriela Barrantes, David H Ackley, Stephanie Forrest, and Darko Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):3–40, 2005.
9. Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. A new CRT-RSA algorithm secure against bellcore attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 311–320. ACM, 2003.
10. Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.
11. Jean-Marie Borello and Ludovic Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.
12. Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the Java Card control flow. In *International Conference on Smart Card Research and Advanced Applications*, pages 283–296. Springer, 2011.
13. Guillaume Bouffard and Jean-Louis Lanet. The ultimate control flow transfer in a Java based smart card. *Computers & Security*, 50:33–46, 2015.
14. Sally C Brailsford, Chris N Potts, and Barbara M Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.
15. Sebanjila K Bukasa, Ronan Lashermes, Jean-Louis Lanet, and Axel Leqay. Let's shock our IoT's heart: ARMv7-M under (fault) attacks. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, page 33. ACM, 2018.
16. Jan Cappaert. *Code obfuscation techniques for software protection*. PhD thesis, University of Katholieke Leuven, 2012.
17. Florence Charreteur and Arnaud Gotlieb. Constraint-based test input generation for Java bytecode. In *IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, pages 131–140. IEEE, 2010.
18. Christian Colberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap Resilient and Stealthy Opaque Constructs. *roc. ymp. Principles of Programming Languages (POPL'98)*, 1998.

19. Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.
20. Stephen Drape. Intellectual property protection using obfuscation. *Proceedings of SAS 2009*, 4779:133–144, 2009.
21. Ninon Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions: reconstruction, analysis and simplification tools*. PhD thesis, University of Paris-Saclay, 2017.
22. Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022, 2015.
23. Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 5(02):56, 2014.
24. Sylvain Guilley, Laurent Sauvage, Jean-Luc Danger, Nidhal Selmane, and Renaud Pacalet. Silicon-level solutions to counteract passive and active attacks. In *FDTC*, pages 3–17. IEEE-CS, 2008.
25. Donald H Habing. The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits. *IEEE Transactions on Nuclear Science*, 39:1647–1653, 1992.
26. Samiya Hamadouche and Jean-Louis Lanet. Virus in a smart card: Myth or reality? *Journal of Information Security and Applications*, 18(2-3):130–137, 2013.
27. Samiya Hamadouche, Mohamed Mezghiche, Arnaud Gotlieb, and Jean-Louis Lanet. Vers une approche de construction de virus pour cartes à puce basée sur la résolution de contraintes. *Actes de la 13 ème édition dAFADL, Atelier Francophone sur les Approches Formelles dans lAssistance au Développement de Logiciels*, 2014.
28. Fred H Hardie and Robert J Suhocki. Design and use of fault simulation for saturn computer design. *IEEE Transactions on Electronic Computers*, (4):412–429, 1967.
29. Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology*, 2018.
30. Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 219–235. Springer, 2013.
31. Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM–software protection for the masses. In *IEEE/ACM 1st International Workshop on Software Protection (SPRO)*, pages 3–9. IEEE, 2015.
32. Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, 2013.
33. Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.
34. Martin S Kelly, Keith Mayes, and John F Walker. Characterising a CPU fault attack model via run-time data analysis. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 79–84. IEEE, 2017.
35. Thomas Korak and Michael Hoefler. On the effects of clock and power supply tampering on two microcontroller platforms. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on*, pages 8–17. IEEE, 2014.
36. Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32, 1992.
37. Julien Lancia. Java Card combined attacks with localization-agnostic fault injection. In *International Conference on Smart Card Research and Advanced Applications*, pages 31–45. Springer, 2012.
38. Douglas Low. Java control flow obfuscation. Master's thesis, University of Auckland, 1998.
39. Matias Madou, Bertrand Anckaert, Bruno De Bus, Koen De Bosschere, Jan Cappaert, and Bart Preneel. On the effectiveness of source code transformations for binary obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP06)*, pages 527–533. CSREA Press, 2006.

40. Premachandran R. Menon and Stephen G. Chappell. Deductive fault simulation with functional blocks. *IEEE Transactions on Computers*, (8):689–695, 1978.
41. Abdelhak Mesbah, Jean-Louis Lanet, and Mohamed Mezghiche. Reverse engineering Java Card and vulnerability exploitation: a shortcut to ROM. *International Journal of Information Security*, pages 1–16, 2017.
42. Abdelhak Mesbah, Mohamed Mezghiche, and Jean-Louis Lanet. Persistent fault injection attack from white-box to black-box. In *5th International Conference on Electrical Engineering Boumerdes (ICEE-B)*, pages 1–6. IEEE, 2017.
43. Ian Miguel and Qiang Shen. Solution techniques for constraint satisfaction problems: Foundations. *Artificial Intelligence Review*, 15(4):243–267, 2001.
44. Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88. IEEE, 2013.
45. Shoei Nashimoto, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki. Buffer overflow attack with multiple fault injection and a proven countermeasure. *Journal of Cryptographic Engineering*, 7(1):35–46, 2017.
46. Oracle. *Java Card^{TM}Platform, Version 3.0.5 Classic Edition : Virtual Machine Specification*. Oracle America, 2015.
47. Roberta Piscitelli, Shivam Bhasin, and Francesco Regazzoni. Fault attacks, injection techniques and tools for simulation. In *Hardware Security and Trust*, pages 27–47. Springer, 2017.
48. Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. Camouflage in malware: from encryption to metamorphism. *International Journal of Computer Science and Network Security*, 12(8):74–83, 2012.
49. Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of ARMv7-M architectures. *arXiv preprint arXiv:1510.01537*, 2015.
50. Jörn-Marc Schmidt and Michael Hutter. *Optical and EM fault-attacks on CRT-based RSA: Concrete results*. na, 2007.
51. Jagsir Singh and Jaswinder Singh. Challenge of Malware Analysis: Malware obfuscation Techniques. *International Journal of Information Security Science*, 7(3):100–110, 2018.
52. Sanjam Singla, Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Detecting and classifying morphed malwares: A survey. *International Journal of Computer Applications*, 122(10), 2015.
53. Sergei P Skorobogatov and Ross J Anderson. Optical fault induction attacks. In *International workshop on cryptographic hardware and embedded systems*, pages 2–12. Springer, 2002.
54. Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM using fault injection. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, 2016.
55. Edward Tsang. *Foundations of constraint satisfaction*. Academic Press Limited, 1995.
56. Ernst G Ulrich, T Baker, and LR Williams. Fault-test analysis techniques based on logic simulation. In *Proceedings of the 9th Design Automation Workshop*, pages 111–115. ACM, 1972.
57. Eric Vetillard and Anthony Ferrari. Combined attacks and countermeasures. In *International Conference on Smart Card Research and Advanced Applications*, pages 133–147. Springer, 2010.
58. David Wagner. Cryptanalysis of a provably secure CRT-RSA algorithm. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 92–97. ACM, 2004.
59. Stefan Winter. *On the Utility of Higher Order Fault Models for Fault Injections*. PhD thesis, Technische Universität, 2015.
60. Hui Xu, Yangfan Zhou, Yu Kang, and Michael R Lyu. On Secure and Usable Program Obfuscation: A Survey. *arXiv preprint arXiv:1710.01139*, 2017.
61. Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA)*, pages 297–300. IEEE, 2010.

62. Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. *Journal of Hardware and Systems Security*, pages 1–20, 2018.